Documentação Completa - Sistema WePayU

Visão Geral

O **WePayU** é um sistema de folha de pagamento desenvolvido em Java que implementa diversos padrões de projeto orientados a objetos. O sistema permite gerenciar empregados de diferentes tipos, processar pagamentos, controlar sindicalização e manter persistência de dados.

Arquitetura do Sistema

Padrão Facade

O sistema utiliza o **padrão Facade** através da classe *Facade*, que simplifica a interface complexa do sistema, fornecendo um ponto de entrada único para todas as operações. A Facade encapsula:

- **EmpregadoService**: Gerenciamento de empregados
- **SindicatoService**: Controle de sindicalização
- **LancamentoService**: Lançamento de cartões e vendas
- **FolhaPagamentoService**: Processamento de folha de pagamento
- **PersistenciaService**: Persistência de dados

Estrutura de Pacotes

```
src/br/ufal/ic/p2/wepayu/
├── commands/          # Padrão Command
├── Exception/         # Tratamento de exceções (53 classes)
├── factories/         # Padrão Factory
├── models/            # Modelos de dados
├── services/          # Camada de serviços
└── utils/             # Utilitários
```

Padrões de Projeto Implementados

Padrão Command

Localização: *src/br/ufal/ic/p2/wepayu/commands/*

O sistema implementa o padrão Command para operações que podem ser desfeitas (undo/redo):

- **Interface Command**: Definir contratos *executar()* e *desfazer()*
- **CommandManager**: Gerenciar pilhas de comandos executados
- **Comandos Concretos**:
    - **CriarEmpregadoCommand**
    - **AlterarEmpregadoCommand**
    - **RemoverEmpregadoCommand**
    - **LancarCartaoCommand**
    - **LancarVendaCommand**
    - **RodaFolhaCommand**
    - **ZerarSistemaCommand**

Benefícios:
- Permite desfazer operações
- Encapsula operações em objetos
- Facilita logging e auditoria

Padrão Memento

Localização: *src/br/ufal/ic/p2/wepayu/commands/*

Implementado para salvar e restaurar estados de objetos:

- **Interface Memento**: Define *restaurar()*
- **Classes Memento Concretas**:
    - **CartaoMemento**
    - **TaxaServicoMemento**
    - **VendaMemento**

Benefícios:
- Preserva estados anteriores
- Suporte a undo/redo
- Isolamento de responsabilidades

Padrão Factory

Localização: *src/br/ufal/ic/p2/wepayu/factories/*

Criação de objetos complexos com validações específicas:

- **EmpregadoFactory**: Cria empregados (assalariados, horistas, comissionados)
- **MembroSindicatoFactory**: Cria membros do sindicato

Benefícios:
- Encapsula lógica de criação
- Centraliza validações
- Facilita manutenção

Padrão Template Method

Localização: *src/br/ufal/ic/p2/wepayu/models/Empregado.java*

A classe abstrata *Empregado* define o template para todos os tipos de empregados:

- Métodos abstratos: *getTipo()*, *getSalario()*
- Métodos concretos: *getNome()*, *getEndereco()*, *ehSindicalizado()*
- Métodos vazios: *lancarCartao()*, *lancarResultadoDeVenda()*

Benefícios:
- Define estrutura comum
- Permite personalização específica
- Evita duplicação de código

Modelos de Dados

Hierarquia de Empregados

Empregado (abstrata)
├── nome: String
├── endereco: String
├── sindicato: MembroSindicato
├── metodoPagamento: MetodoPagamento
└── agendaPagamento: AgendaPagamento

+ getTipo(): String (abstrato)
+ getSalario(): String (abstrato)
+ ehSindicalizado(): String
+ lancarCartao(CartaoDePonto): void

+ lancarResultadoDeVenda(ResultadoDeVenda): void

```
├── EmpregadoAssalariado
│   ├── salarioMensal: double
│   └── getTipo(): "assalariado"

├── EmpregadoHorista
│   ├── salarioPorHora: double
│   ├── cartoes: ArrayList<CartaoDePonto>
│   └── getTipo(): "horista"

└── EmpregadoComissionado
    ├── salarioMensal: double
    ├── taxaDeComissao: double
    ├── resultadoDeVenda: ArrayList<ResultadoDeVenda>
    └── getTipo(): "comissionado"
```

Características por Tipo

EmpregadoAssalariado

- Salário fixo mensal
- Pagamento independente de horas
- Agenda padrão: mensal

EmpregadoHorista

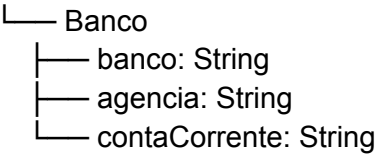
- Salário por hora trabalhada
- Cartões de ponto para registro
- Horas normais (≤8h) e extras (>8h)
- Agenda padrão: semanal

EmpregadoComissionado

- Salário base + comissão sobre vendas
- Registro de vendas realizadas
- Agenda padrão: bi-semanal

Métodos de Pagamento

```
MetodoPagamento (interface)
├── EmMaos
├── Correios
```



Funcionalidades Principais

Gerenciamento de Empregados

- **Criação**: Empregados com validações específicas por tipo
- **Alteração**: Modificação de atributos com validações
- **Remoção**: Exclusão com verificação de existência
- **Consulta**: Busca de informações detalhadas

Sistema de Sindicato

- **Criação de Membros**: Associação de empregados ao sindicato
- **Taxas Sindicais**: Desconto mensal automático
- **Taxas de Serviço**: Lançamento de taxas adicionais
- **Cálculo de Descontos**: Total automático de taxas

Lançamentos

- **Cartões de Ponto**: Para empregados horistas
- **Vendas**: Para empregados comissionados
- **Taxas de Serviço**: Para membros do sindicato

Folha de Pagamento

- **Cálculo Automático**: Salários baseados em tipo de empregado
- **Geração de Arquivos**: Folhas salvas em formato texto
- **Agendas Flexíveis**: Suporte a diferentes períodos de pagamento

Agendas de Pagamento

- **Padrão**: semanal 5, semanal 2 5, mensal $
- **Customizadas**: Criação de agendas personalizadas
- F**ormatos Suportados**:
    - *semanal X*: Pagamento semanal no dia X
    - *semanal X Y*: Pagamento a cada X semanas no dia Y
    - *mensal X*: Pagamento mensal no dia X

- *mensal $*: Pagamento no último dia do mês

Persistência

- **Salvamento Automático**: Após cada operação
- **Carregamento**: Dados restaurados na inicialização
- **Arquivos XML**: *empregados.xml*, *sindicato.xml*, *agendas.xml*

Sistema de Undo/Redo

- **Desfazer**: Última operação executada
- **Refazer**: Operação anteriormente desfeita
- **Histórico**: Pilhas de comandos executados

Principais Métodos e Lógica de Implementação

Métodos da Facade (Interface Principal)

Gerenciamento de Empregados

criarEmpregado(nome, endereco, tipo, salario)

Funcionalidade: Cria empregados assalariados e horistas
Lógica:

- Valida parâmetros obrigatórios (nome, endereco, tipo, salario)
- Usa *EmpregadoFactory* para criar instância apropriada
- Gera ID único sequencial
- Cria *CriarEmpregadoCommand* e executa via *CommandManager*
- Salva automaticamente no sistema de persistência

criarEmpregado(nome, endereco, tipo, salario, comissao)

Funcionalidade: Cria empregados comissionados
Lógica:

- Valida parâmetros incluindo comissão
- Cria *EmpregadoComissionado* via Factory
- Define agenda padrão "semanal 2 5" para comissionados
- Registra comando para permitir undo/redo

alteraEmpregado(emp, atributo, valor)

Funcionalidade: Altera atributos de empregados existentes

Lógica:

- Valida existência do empregado
- Cria *AlterarEmpregadoCommand* com memento do estado anterior
- Executa alteração via CommandManager
- Suporta alteração de: nome, endereco, tipo, salario, metodoPagamento, sindicalizado

getAtributoEmpregado(emp, atributo)

Funcionalidade: Consulta atributos específicos de empregados
Lógica:

- Busca empregado no mapa por ID
- Retorna valor formatado do atributo solicitado
- Suporta consulta de dados bancários, sindicais e de comissão

Sistema de Lançamentos

lancaCartao(emp, data, horas)

Funcionalidade: Registra cartão de ponto para empregados horistas
Lógica:

- Valida se empregado existe e é horista
- Valida formato de data (dd/MM/yyyy) e valor das horas
- Cria *LancarCartaoCommand* com memento do estado anterior
- Adiciona cartão à lista do empregado horista
- Executa via CommandManager para permitir undo

lancaVenda(emp, data, valor)

Funcionalidade: Registra venda para empregados comissionados
Lógica:

- Valida se empregado existe e é comissionado
- Valida formato de data e valor monetário
- Cria *LancarVendaCommand* com memento
- Adiciona venda à lista do empregado comissionado

getHorasNormaisTrabalhadas(emp, dataInicial, dataFinal)

Funcionalidade: Calcula horas normais (≤ 8h) em período
Lógica:

- Filtra cartões dentro do período especificado
- Aplica *Math.min(cartao.getHoras(), 8.0)* para cada cartão
- Soma todas as horas normais
- Retorna valor formatado com vírgula decimal

getHorasExtrasTrabalhadas(emp, dataInicial, dataFinal)

Funcionalidade: Calcula horas extras (> 8h) em período
Lógica:

- Filtra cartões no período
- Aplica *Math.max(cartao.getHoras() - 8.0, 0)* para cada cartão
- Soma todas as horas extras

Sistema de Folha de Pagamento

totalFolha(data)

Funcionalidade: Calcula total da folha para uma data específica
Lógica:

- Itera sobre todos os empregados
- Verifica se cada empregado deve receber na data (via agenda)
- Calcula salário específico por tipo de empregado
- Soma todos os valores com precisão BigDecimal
- Retorna total formatado em moeda brasileira

rodaFolha(data, arquivo)

Funcionalidade: Processa e gera arquivo de folha de pagamento
Lógica:

- Agrupa empregados por tipo (horista, assalariado, comissionado)
- Ordena empregados por nome
- Calcula salários, descontos e salário líquido para cada um
- Gera arquivo formatado com totais por categoria
- Registra comando para permitir undo

Cálculo de Salários por Tipo

Empregados Horistas

```
// Cálculo de horas normais e extras
horasNormais = Math.min(horasCartao, 8.0)
horasExtras = Math.max(horasCartao - 8.0, 0)

// Salário bruto
salarioBruto = (horasNormais × salarioHora) + (horasExtras × salarioHora × 1.5)
```

Empregados Assalariados

```
// Salário fixo mensal
salarioBruto = salarioMensal
```

Empregados Comissionados

// Salário base quinzenal
salarioBase = (salarioMensal × 12) / 26

// Comissão sobre vendas do período
comissao = totalVendas × taxaComissao

// Salário bruto
salarioBruto = salarioBase + comissao

Sistema de Descontos

Descontos para Horistas (Sistema de Dívida)

- **Dívida Sindical Acumulada**: 7 dias de taxa por pagamento semanal
- **Taxas de Serviço**: Do período atual (última semana)
- **Lógica Especial**: Se salário líquido < 0, ajusta dívida e limita descontos

Descontos para Assalariados

- **Taxa Sindical**: Dias do mês × taxa diária
- **Taxas de Serviço**: Do período mensal

Descontos para Comissionados

- **Taxa Sindical**: 14 dias × taxa diária
- **Taxas de Serviço**: Do período quinzenal

Sistema de Undo/Redo

undo()

Funcionalidade: Desfaz última operação executada
Lógica:

- Verifica se sistema não foi encerrado
- Remove comando da pilha de histórico
- Executa método *desfazer()* do comando
- Adiciona comando à pilha de redo

redo()

Funcionalidade: Refaz última operação desfeita
Lógica:

- Remove comando da pilha de redo
- Executa método *executar()* do comando
- Adiciona comando de volta à pilha de histórico

Persistência de Dados

salvarSistema()

Funcionalidade: Salva todos os dados em arquivos XML
Lógica:

- Serializa mapa de empregados para *empregados.xml*
- Serializa mapa de membros sindicato para *sindicato.xml*
- Serializa agendas customizadas para *agendas.xml*
- Usa *XMLEncoder* para serialização Java Beans

carregarSistema()

Funcionalidade: Carrega dados salvos na inicialização
Lógica:

- Deserializa arquivos XML usando *XMLDecoder*
- Restaura mapas de empregados e membros sindicato
- Restaura contador de IDs sequenciais
- Inicializa agendas customizadas

Padrões de Projeto em Ação

Command Pattern

- **Criação**: *CriarEmpregadoCommand* encapsula operação de criação
- **Execução**: *commandManager.executar(command)* executa e armazena
- **Undo**: *command.desfazer()* restaura estado anterior via memento

Memento Pattern

- **CartaoMemento**: Salva lista de cartões antes de adicionar novo
- **VendaMemento**: Salva lista de vendas antes de adicionar nova
- **Restauração**: Comandos usam mementos para desfazer operações

Factory Pattern

- **EmpregadoFactory**: Centraliza criação com validações específicas
- **Validações**: Salário numérico, comissão não-negativa, tipos válidos
- **Polimorfismo**: Retorna instância apropriada baseada no tipo

Template Method Pattern

- **Empregado**: Classe abstrata com métodos abstratos *getTipo(), getSalario()*
- **Subclasses**: Implementam comportamento específico por tipo
- **Métodos Concreto**s: *getNome(), getEndereco()* compartilhados

Facade Pattern

- I**nterface Simplificada**: Cliente interage apenas com *Facade*
- **Encapsulamento**: Oculta complexidade dos 5 serviços internos
- **Coordenação**: Facade orquestra chamadas entre serviços

Tratamento de Exceções

O sistema possui um sistema robusto de exceções com **53 classes específicas**:

Categorias de Exceções

- **Validação de Dados**: *NomeNaoPodeSerNuloException*, *SalarioDeveSerNumericoException*, etc
- **Empregados**: *EmpregadoNaoExisteException*, *EmpregadoNaoEhHoristaException*, etc
- **Sindicato**: *IdentificacaoSindicatoJaExisteException*, etc
- **Sistema**: *NaoPodeComandosAposEncerrarSistemaException*, etc

Testes

O sistema utiliza o framework **EasyAccept** para testes automatizados:

Casos de Uso Testados

- **US1**: Criação de empregados
- **US2**: Remoção de empregados
- **US3**: Lançamento de cartões de ponto
- **US4**: Lançamento de vendas
- **US5**: Lançamento de taxas de serviço
- **US6**: Alterar detalhes de um empregado
- **US7**: Rodar folha de pagamento
- **US8**: Undo/Redo
- **US9**: Agenda de pagamento
- **US10**: Criação de novas agendas de pagamento

Estrutura de Arquivos

Arquivos de Dados

- *empregados.xml*: Dados dos empregados
- *sindicato.xml*: Dados do sindicato
- *agendas.xml*: Agendas customizadas

Arquivos de Folha

- *folha-YYYY-MM-DD.txt*: Folhas de pagamento geradas
- *ok/*: Diretório com folhas de referência

Arquivos de Teste

- *tests/us*.txt*: Casos de uso para testes
- *lib/easyaccept.jar*: Framework de testes

Fluxo de Execução

1. Main.main() → EasyAccept
2. EasyAccept → Facade
3. Facade → Services específicos
4. Services → Models/Factories
5. CommandManager → Commands
6. PersistenciaService → XML files

Qualidades do Código

Pontos Fortes

1. **Separação de Responsabilidades**: Cada classe tem uma responsabilidade específica
2. **Reutilização**: Padrões de projeto facilitam reutilização
3. **Extensibilidade**: Fácil adição de novos tipos de empregados
4. **Manutenibilidade**: Código bem estruturado e documentado
5. **Testabilidade**: Sistema de testes abrangente

Padrões de Design Aplicados

- **SOLID Principles**: Single Responsibility, Open/Closed, Liskov Substitution
- **DRY**: Don't Repeat Yourself
- **Encapsulation****: Dados protegidos por métodos
- **Polymorphism****: Uso de herança e interfaces

Métricas do Projeto

- **53 Classes de Exceção**: Tratamento específico de erros
- **10 Casos de Uso Testados**: Cobertura completa de funcionalidades
- **5 Padrões de Projeto (contando com Facade)**: Implementação de boas práticas
- **3 Tipos de Empregados**: Flexibilidade no modelo de negócio
- **3 Métodos de Pagamento**: Diversidade de opções
- **Sistema de Undo/Redo**: Funcionalidade avançada

Impacto Educacional

O projeto WePayU serve como um excelente exemplo de:

- **Aplicação Prática de Padrões de Projeto**: Demonstra como implementar padrões em sistemas reais
- **Desenvolvimento Orientado a Objetos**: Mostra boas práticas de POO
- **Tratamento Robusto de Exceções**: Sistema abrangente de tratamento de erros
- **Estruturação de Código**: Organização para manutenibilidade
- **Implementação de Testes**: Testes automatizados abrangentes

Conclusão

O sistema WePayU demonstra uma implementação robusta de padrões de projeto orientados a objetos, oferecendo:

- **Flexibilidade**: Fácil extensão e modificação
- **Robustez**: Tratamento abrangente de exceções
- **Usabilidade**: Interface simplificada via Facade
- **Confiabilidade**: Sistema de testes automatizados
- **Persistência**: Salvamento automático de dados

Este projeto serve como um excelente exemplo de como aplicar padrões de projeto para criar sistemas de software bem estruturados, demonstrando a importância de boas práticas de desenvolvimento orientado a objetos.

O WePayU não é apenas um sistema funcional, mas também um exemplo educacional valioso que mostra como conceitos teóricos podem ser aplicados na prática para criar soluções robustas e escaláveis.